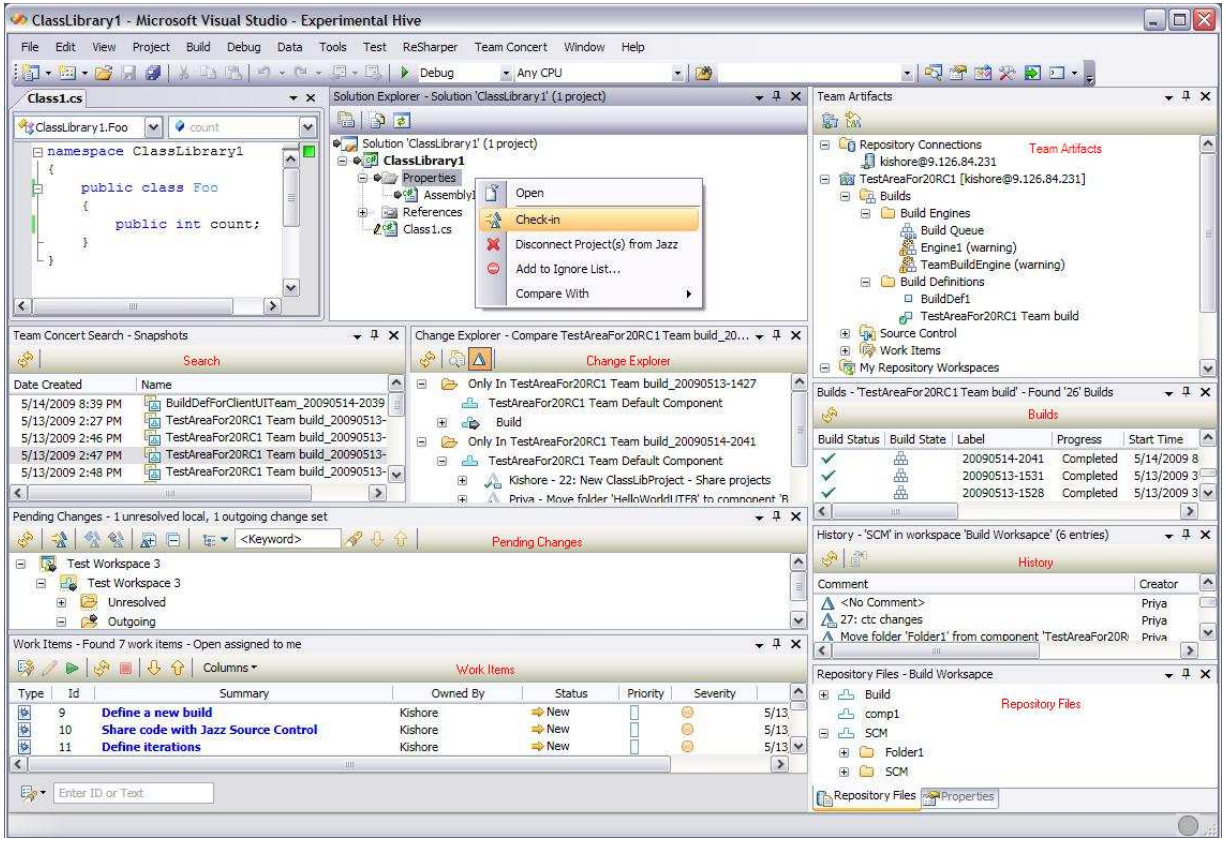
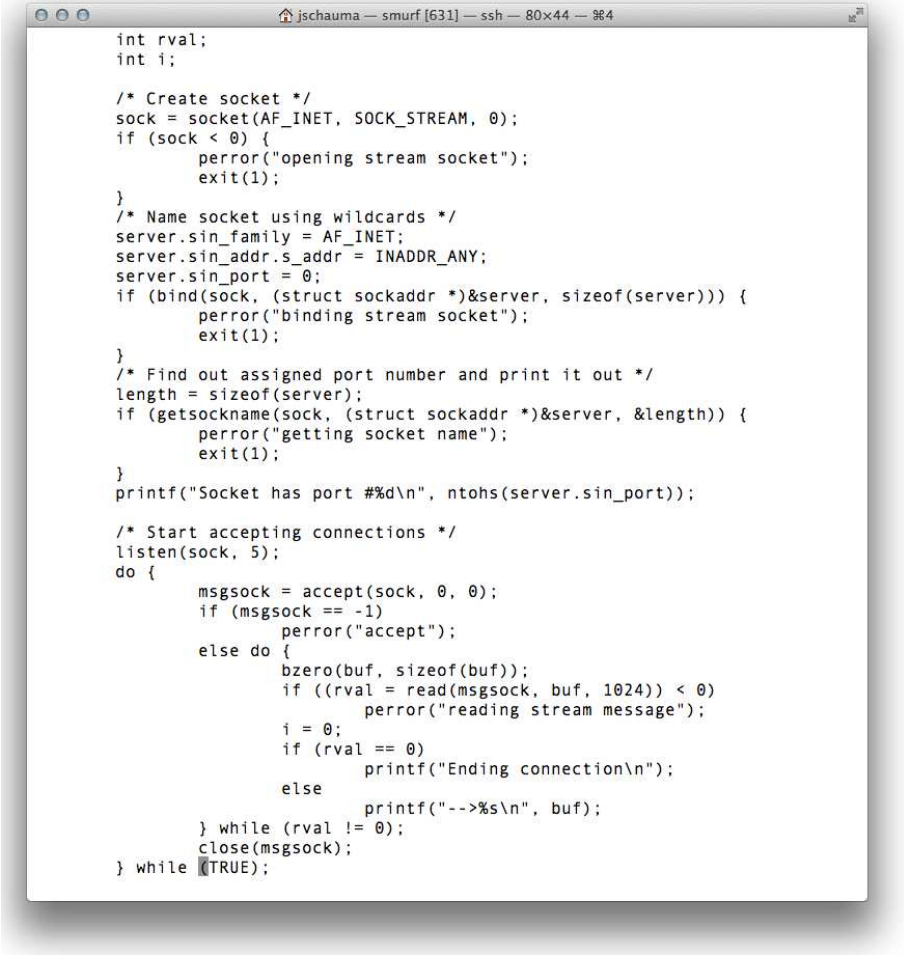
# CS631 - Advanced Programming in the UNIX Environment
# –
# UNIX development tools

Department of Computer Science
Stevens Institute of Technology
Jan Schaumann
jschauma@stevens.edu
https://stevens.netmeister.org/631/

# Software Development Tools



# Software Development Tools



# Software Development Tools

UNIX Userland is an IDE – essential tools that follow the paradigm of "Do one thing, and do it right" can be combined.

The most important tools are:

- `$EDITOR`
- the compiler toolchain
- `gdb(1)` – debugging your code
- `make(1)` – project build management, maintain program dependencies
- `diff(1)` and `patch(1)` – report and apply differences between files
- `cvs(1)`, `svn(1)`, `git(1)` etc. – distributed project management, version control

# EDITOR

Know your `$EDITOR`. Core functionality:

- syntax highlighting
- efficient keyboard maneuvering
- setting markers, using buffers
- copy, yank, fold e.g. blocks
- search and replace
- window splitting
- autocompletion
- jump to definition / manual page
- applying external commands and filters

# EDITOR

Examples given using `vim(1)`.

Efficient keyboard maneuvering:

- h, j, k, l
- w, b, e
- /, ?, ^, $
- ^D, ^B
- zz, zt, zb
- %, ]}, [{
- :n, :prev, :rew

# EDITOR

Examples given using `vim(1)`.

Copy, yank, fold, markers, buffers etc.:

- m [a-zA-Z]
- :marks
- v, V
- =
- d, y, ”xy, ”xp
- zf, zA

# EDITOR

Examples given using `vim(1)`.

Look-ups:

- `find /usr/src -name '*[ch]' -print | exec ctags -f ~/.ctgs`
- `echo "set tags+=~/.ctags" >> ~/.vimrc`
- Ctrl+], Ctrl+t – jump to definition and back
- K – jump to manual page
- Ctrl+N – autocomplete

# EDITOR

Examples given using `vim(1)`.

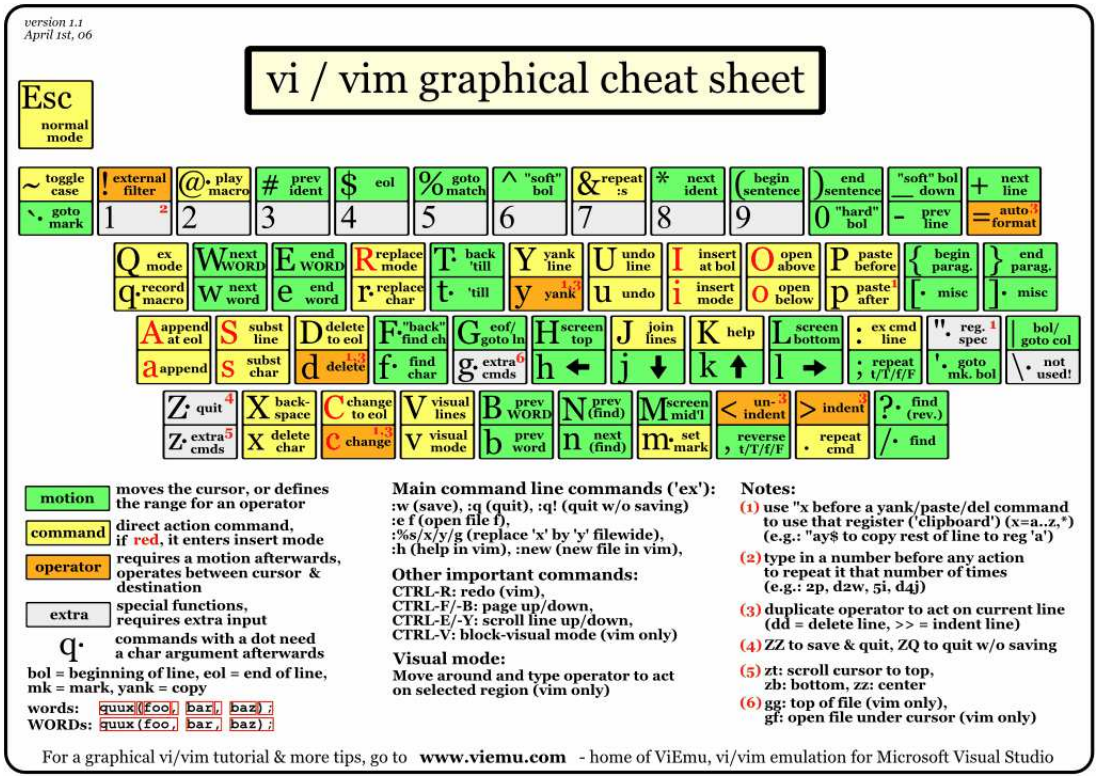
Integration with compiler, debugger, `make(1)` etc.

```
vim welcome.c
:make
Ctrl+]
:cnext
...
```

Finally, two of your most powerful Unix IDE integrations are a terminal multiplexer (e.g. `screen(1)` or `tmux(1)`) and copious use of Ctrl+Z (i.e., the shell's job control mechanisms).

# EDITOR

Examples given using `vim(1)`.



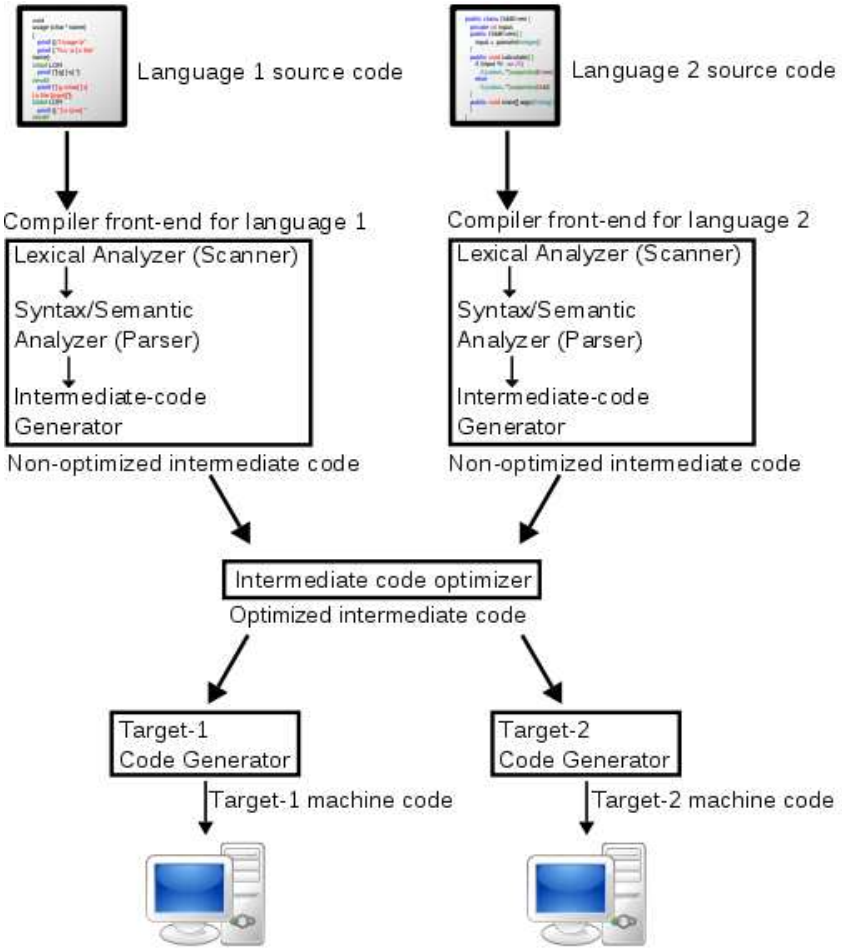

`https://duckduckgo.com/?q=vim+tutorial`

# Compilers

A compiler translates *source code* from a high-level programming language into *machine code* for a given architecture by performing a number of steps:

- lexical analysis
- preprocessing
- parsing
- semantic analysis
- code optimization
- code generation
- assembly
- linking

# Compilers


Language 1 source code
Language 2 source code
Compiler front-end for language 1
Lexical Analyzer (Scanner)
Syntax/Semantic Analyzer (Parser)
Intermediate-code Generator
Non-optimized intermediate code
Compiler front-end for language 2
Lexical Analyzer (Scanner)
Syntax/Semantic Analyzer (Parser)
Intermediate-code Generator
Non-optimized intermediate code
Intermediate code optimizer
Optimized intermediate code
Target-1 Code Generator
Target-2 Code Generator
Target-1 machine code
Target-2 machine code

# Compilers

There are many different closed- and open-source compiler chains:

- Intel C/C++ Compiler (or `icc`)
- Turbo C / Turbo C++ / C++Builder (Borland)
- Microsoft Visual C++
- ...

- Clang (a frontend to LLVM)
- GNU Compiler Collection (or `gcc`)
- Portable C Compiler (or `pcc`)
- ...

# The compiler toolchain

The compiler chain or driver usually performs preprocessing (e.g. via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

# Preprocessing

The compiler chain or driver usually performs preprocessing (e.g. via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

```
$ cd compilechain
$ cat hello.c
$ man cpp
$ cpp hello.c hello.i
$ file hello.i
$ man cc
$ cc -v -E hello.c > hello.i
$ more hello.i
$ cc -v -DFOOD=\"Avocado\" -E hello.c > hello.i.2
$ diff -bu hello.i hello.i.2
```

# Compilation

The compiler chain or driver usually performs preprocessing (e.g. via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

```
$ more hello.i
$ cc -v -S hello.i
$ file hello.s
$ more hello.s
```

# Assembly

The compiler chain or driver usually performs preprocessing (e.g. via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

```
$ as -o hello.o hello.s
$ file hello.o
$ cc -v -c hello.s
$ objdump -d hello.o
[...]
```

# Linking

The compiler chain or driver usually performs preprocessing (e.g. via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

```
$ ld hello.o
[...]
$ ld hello.o -lc
[...]
$ cc -v hello.o
[...]
$ ld -dynamic-linker /usr/libexec/ld.elf_so                    \
        /usr/lib/crt0.o /usr/lib/crti.o /usr/lib/crtbegin.o \
        hello.o -lc /usr/lib/crtend.o /usr/lib/crtn.o
$ file a.out
$ ./a.out
```

# Linking

The compiler chain or driver usually performs preprocessing (e.g. via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

```
$ cc -v -DFOOD=\"Avocado\" hello.c 2>&1 | more
```

# `cc(1)` and `ld(1)`

The compiler chain or driver usually performs preprocessing (e.g. via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

Different flags can be passed to `cc(1)` to be passed through to each tool as well as to affect all tools.

```
$ cc -v -O2 -g hello.c 2>&1 | more
```

# `cc(1)` and `ld(1)`

The compiler chain or driver usually performs preprocessing (e.g. via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

Different flags can be passed to `cc(1)` to be passed through to each tool as well as to affect all tools.

The order of the command line flags *may* play a role! Directories searched for libraries via `-L` and the resolving of undefined symbols via `-l` are examples of position sensitive flags.

```
$ cc -v main.c -L./lib2 -L./lib -lldtest 2>&1 | more
```

```
$ cc -v main.c -L./lib -L./lib2 -lldtest 2>&1 | more
```

# cc(1) and ld(1)

The compiler chain or driver usually performs preprocessing (e.g. via cpp(1)), compilation (cc(1)), assembly (as(1)) and linking (ld(1)).

Different flags can be passed to cc(1) to be passed through to each tool as well as to affect all tools.

The order of the command line flags *may* play a role! Directories searched for libraries via -L and the resolving of undefined symbols via -l are examples of position sensitive flags.

The behavior of the compiler toolchain may be influenced by environment variables (eg TMPDIR, SGI_ABI) and/or the compilers default configuration file (MIPSPro's /etc/compiler.defaults or gcc's specs).

```
$ cc -v hello.c
$ TMPDIR=/var/tmp cc -v hello.c
$ cc -dumpspec
```

# A Debugger

# gdb(1)

The purpose of a debugger such as `gdb(1)` is to allow you to see what is going on "inside" another program while it executes – or what another program was doing at the moment it crashed. `gdb` allows you to

- make your program stop on specified conditions (for example by setting *breakpoints*)
- examine what has happened, when your program has stopped (by looking at the *backtrace*, inspecting the value of certain variables)
- inspect control flow (for example by *stepping* through the program)

Other interesting things you can do:

- examine stack frames: *info frame*, *info locals*, *info args*
- examine memory: *x*
- examine assembly: *disassemble func*

# gdb(1)

```
$ cd gdb-examples/ls
$ ./a.out
Memory fault (core dumped)
$ gdb ./a.out
(gdb) run ~/tmp

Program received signal SIGSEGV, Segmentation fault.
0x0000000000401a6d in record_stat
(statp=0x7f7fffffdaf0, path_name=0x7f7ff7b09175 "noown")
    at ls.c:497
497             strcpy(new_node->owner_name, password->pw_name);

(gdb) bt
(gdb) frame 0
(gdb) li
(gdb) print password
```

# gdb(1)

```
(gdb) start -l
[...]
(gdb) watch blocksize
[...]
(gdb) c
[...]
(gdb) li
[...]
(gdb) show environment BLOCKSIZE
[...]
(gdb) call strtoll(blocksize_str, 0, 0)
[...]
```

# make(1)

# make(1)

make(1) is a command generator and build utility. Using a description file (usually *Makefile*) it creates a sequence of commands for execution by the shell.

- used to sort out dependency relations among files

# `make(1)`

`make(1)` is a command generator and build utility. Using a description file (usually *Makefile*) it creates a sequence of commands for execution by the shell.

- used to sort out dependency relations among files
- avoids having to rebuild the entire project after modification of a single source file

# `make(1)`

`make(1)` is a command generator and build utility. Using a description file (usually *Makefile*) it creates a sequence of commands for execution by the shell.

- used to sort out dependency relations among files
- avoids having to rebuild the entire project after modification of a single source file
- performs *selective* rebuilds following a *dependency graph*

# `make(1)`

`make(1)` is a command generator and build utility. Using a description file (usually *Makefile*) it creates a sequence of commands for execution by the shell.

- used to sort out dependency relations among files
- avoids having to rebuild the entire project after modification of a single source file
- performs *selective* rebuilds following a *dependency graph*
- allows simplification of rules through use of *macros* and *suffixes*, some of which are internally defined

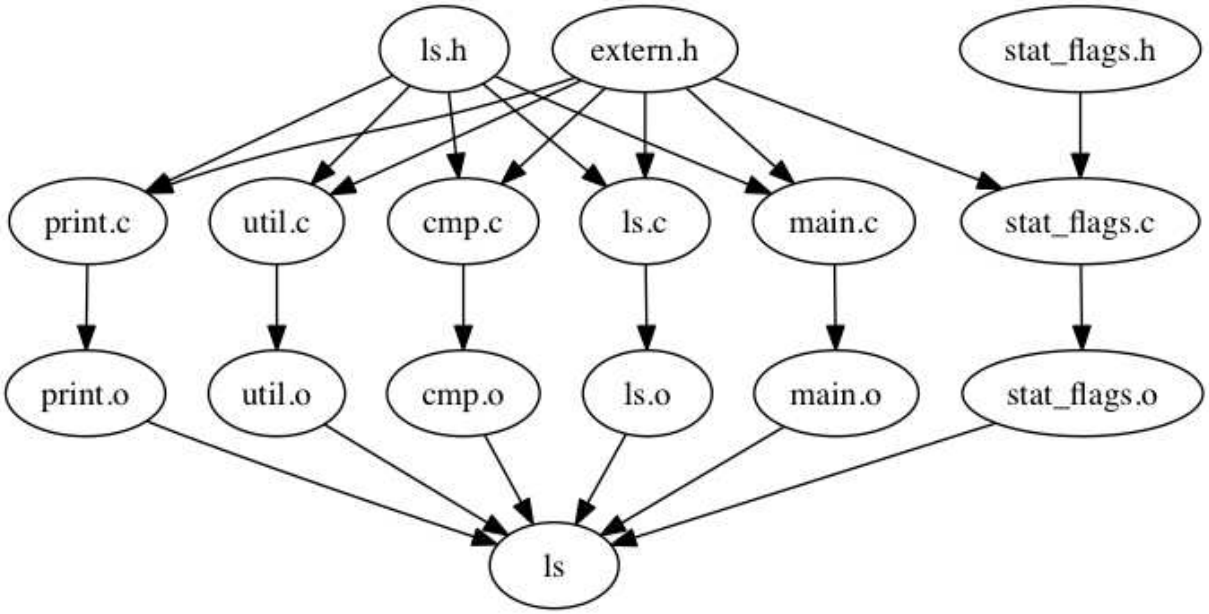
# `make(1)`

`make(1)` is a command generator and build utility. Using a description file (usually *Makefile*) it creates a sequence of commands for execution by the shell.

- used to sort out dependency relations among files
- avoids having to rebuild the entire project after modification of a single source file
- performs *selective* rebuilds following a *dependency graph*
- allows simplification of rules through use of *macros* and *suffixes*, some of which are internally defined
- different versions of `make(1)` (BSD make, GNU make, Sys V make, ...) may differ (among other things) in
  - variable assignment and expansion/substitution
  - including other files
  - flow control (for-loops, conditionals etc.)

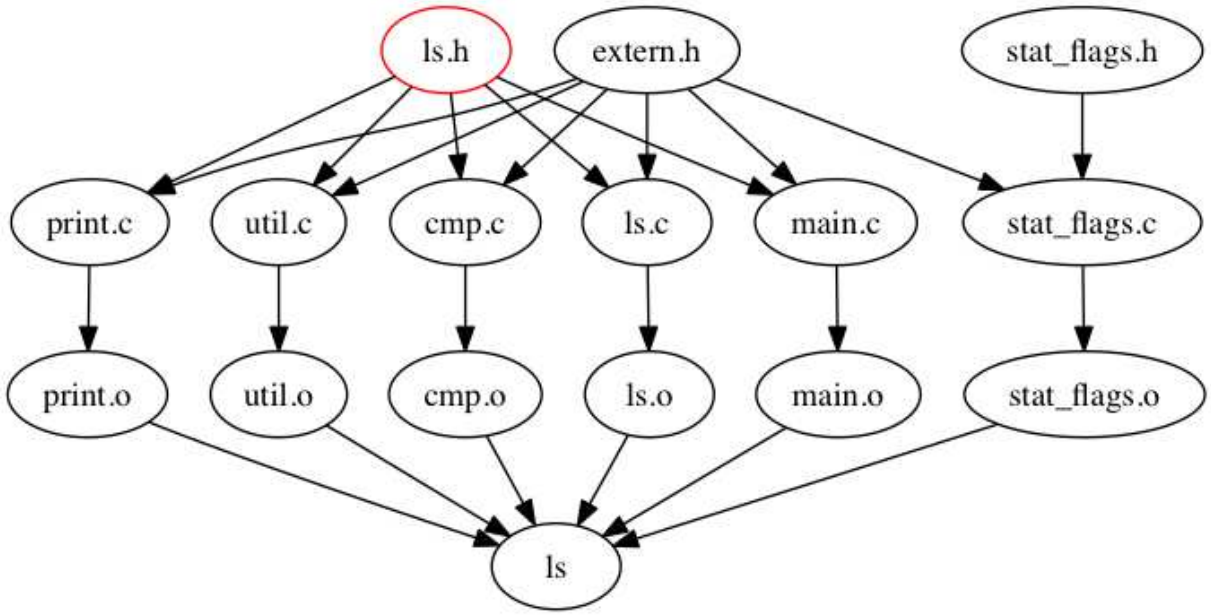
# make(1)

```
$ cd make-examples
$ ls *.[ch]
cmp.c       ls.c        main.c        stat_flags.c        util.c
extern.h    ls.h        print.c       stat_flags.h
```



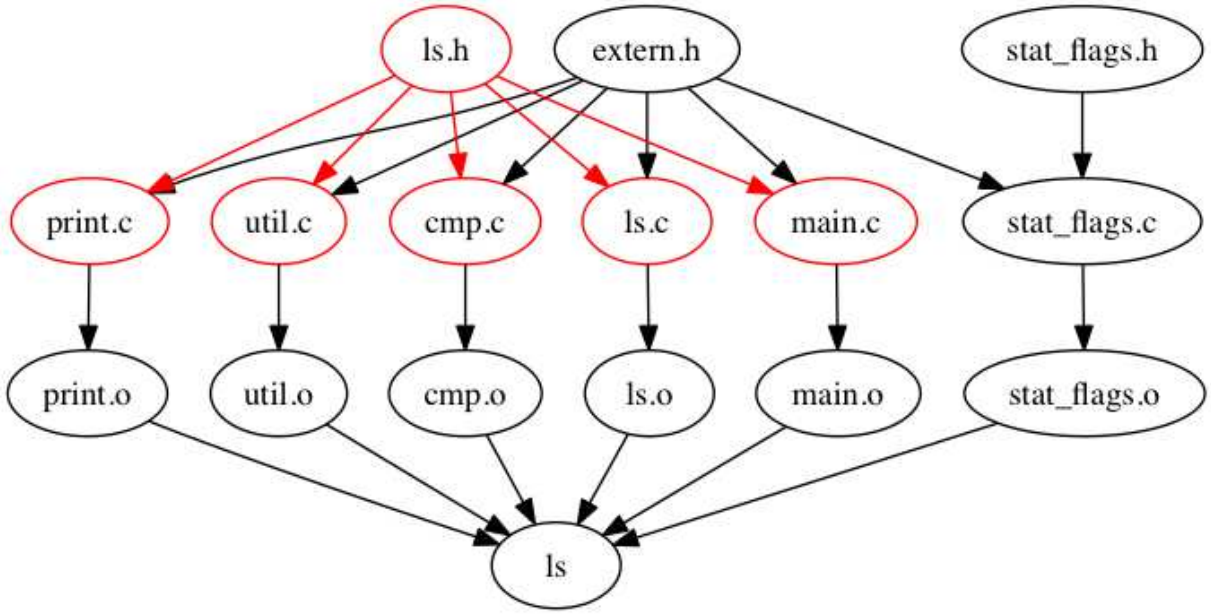
# make(1)

```
$ cd make-examples
$ ls *.[ch]
cmp.c       ls.c        main.c        stat_flags.c        util.c
extern.h    ls.h        print.c       stat_flags.h
```



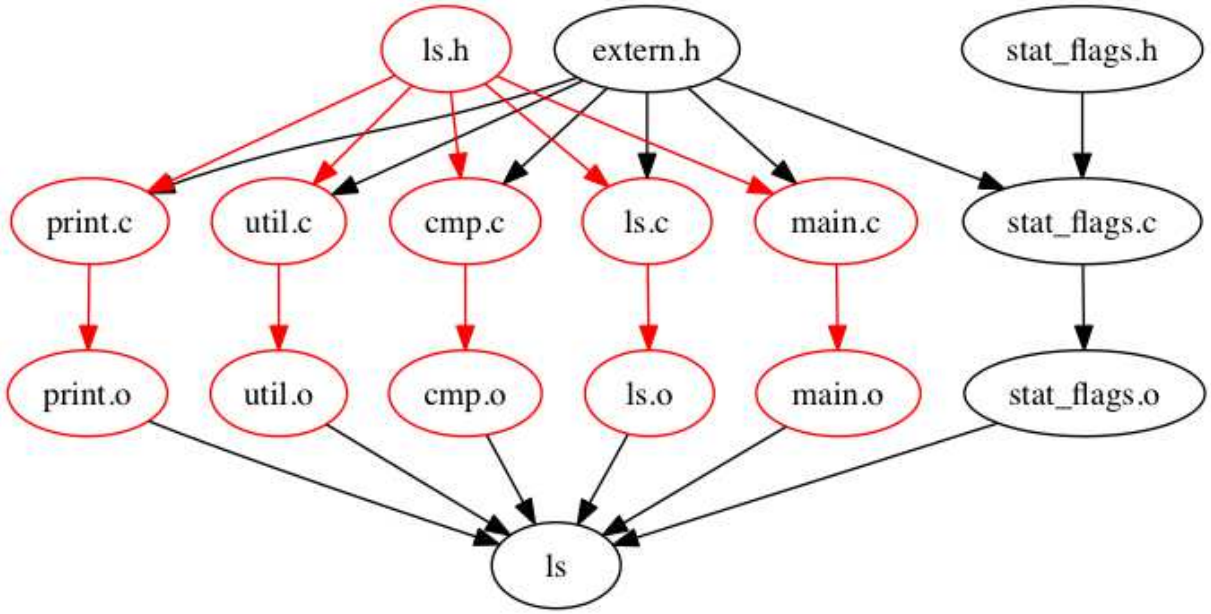
# make(1)

```
$ cd make-examples
$ ls *.[ch]
cmp.c       ls.c        main.c        stat_flags.c        util.c
extern.h    ls.h        print.c       stat_flags.h
```



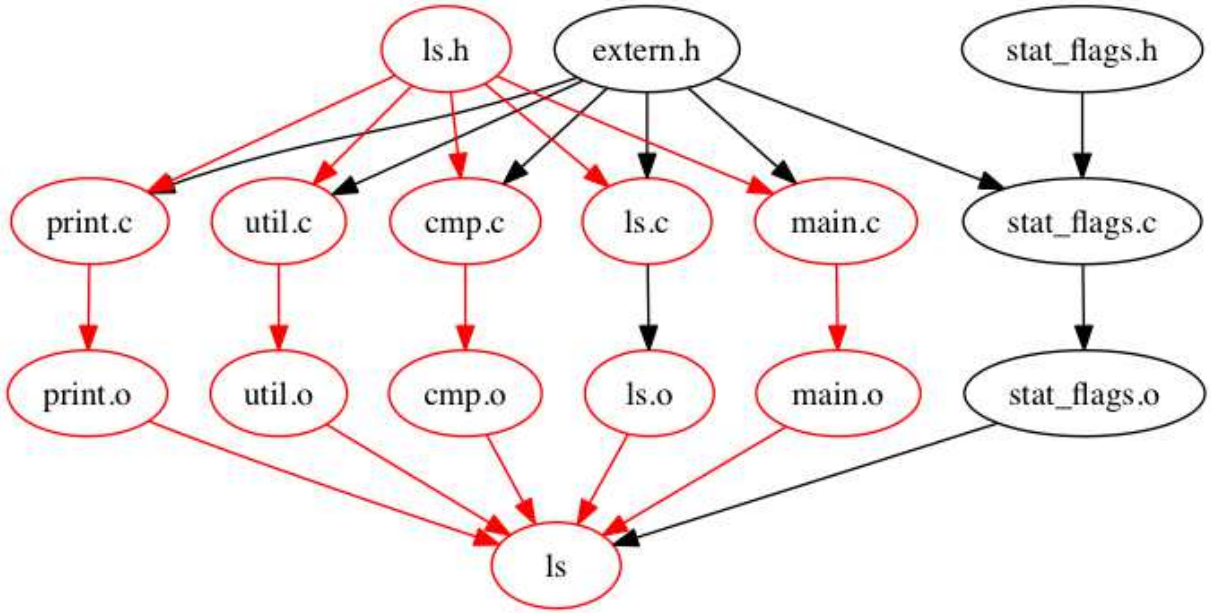
# make(1)

```
$ cd make-examples
$ ls *.[ch]
cmp.c       ls.c        main.c        stat_flags.c        util.c
extern.h    ls.h        print.c       stat_flags.h
```



# make(1)

```
$ cd make-examples
$ ls *.[ch]
cmp.c       ls.c        main.c        stat_flags.c        util.c
extern.h    ls.h        print.c       stat_flags.h
```



# make(1)

```
$ ln -s Makefile.1 Makefile
$ make # or: make -f Makefile.1
[...]
$ make
```

# make(1)

```
$ ln -sf Makefile.2 Makefile
$ make # or: make -f Makefile.2
[...]
$ make clean
$ export CFLAGS="-Wall -Werror"
$ make
[...]
$ make clean
[...]
```

# make(1)

```
$ make -f Makefile.3
[...]
$ make -f Makefile.3 ls.txt
[...]
$
```

# make(1)

```
$ ln -sf Makefile.4 Makefile
$ make help
[...]
$ make showvars
[...]
$ make CFLAGS="${CFLAGS}" showvars
[...]
```

# Priority of Macro Assignments for `make(1)`

1. Internal (default) definitions of `make(1)`
2. Current shell environment variables. This includes macros that you enter on the *make* command line itself.
3. Macro definitions in *Makefile*.
4. Macros entered on the `make(1)` command line, if they follow the *make* command itself.

# Ed is the standard text editor.

```
$ ed
?
help
?
quit
?
exit
?
bye
?
eat flaming death
?
^C
?
^D
?
```

# Ed is the standard text editor.

```
$ ed
a
ed is the standard Unix text editor.
This is line number two.
.
2i

.
%l
3s/two/three/
w foo
q
$ cat foo
```

# `diff(1)` and `patch(1)`

`diff(1)`:

- compares files line by line
- output may be used to automatically edit a file
- can produce human “readable” output as well as diff entire directory structures
- output called a *patch*

# `diff(1)` and `patch(1)`

`patch(1)`:

- applies a `diff(1)` file (aka *patch*) to an original
- may back up original file
- may guess correct format
- ignores leading or trailing “garbage”
- allows for reversing the patch
- may even correct context line numbers

# diff(1) and patch(1)

```
$ diff Makefile.2 Makefile.5
[...]
$ cp Makefile.2 /tmp
$ ( diff -e Makefile.2 Makefile.5; echo w; ) | ed Makefile.2
$ diff Makefile.[25]
$ mv /tmp/Makefile.2 .
$ diff -c Makefile.[25]
$ diff -u Makefile.[25] > /tmp/patch
$ patch </tmp/patch
$ diff Makefile.[25]
```

# `diff(1)` and `patch(1)`

Difference in `ls(1)` between NetBSD and OpenBSD:

```
$ diff -bur netbsd/src/bin/ls openbsd/src/bin/ls
```

Difference in `ls(1)` between NetBSD and FreeBSD:

```
$ diff -bur netbsd/src/bin/ls freebsd-ls/ls
```

# Revision Control

Version control systems allow you to

- collaborate with others
- simultaneously work on a code base
- keep old versions of files
- keep a log of the who, when, what, and why of any changes
- perform release engineering by creating *branches*

# Revision Control

- Source Code Control System (*SSCS*) begat the Revision Control System (*RCS*).
- RCS operates on a single file; still in use for misc. OS config files
- the Concurrent Versions System (*CVS*) introduces a client-server architecture, control of hierarchies
- *Subversion* provides atomic commits, renaming, cheap branching etc.
- *Git*, *Mercurial* etc. implement a *distributed* approach (ie peer-to-peer versus client-server), adding other features (cryptographic authentication of history, ...)

# Revision Control

Examples:

`http://cvsweb.netbsd.org/bsdweb.cgi/src/bin/ls/`

`http://svnweb.freebsd.org/base/stable/9/bin/ls/`

`http://git.savannah.gnu.org/cgit/coreutils.git/log/`

`http://cvsweb.netbsd.org/bsdweb.cgi/src/share/misc/bsd-family-tree?rev=HEAD`

# Revision Control: Branching

Different strategies:

- trunk / master is fragile
  - *trunk* is work in progress, may not even compile
  - all work happens in *trunk*
  - releases are tagged on *trunk*, then branched
- trunk / master is stable
  - *master* is always stable
  - all work is done in branches (feature or bugfix)
  - feature branches are deleted after merge
  - releases are made automatically from master

You may combine these as *release branching* / *feature branching* / *task branching*.

# Commit Messages

Commit messages are like comments: too often useless and misleading, but critical in understanding human thinking behind the code.

Commit messages should be full sentences in correct and properly formatted English.

Commit messages briefly summarize the *what*, but provide important historical context as to the *how* and, more importantly, *why*.

Commit messages SHOULD reference and integrate with ticket tracking systems.

See also:

- `http://is.gd/Wd1LhA`
- `http://is.gd/CUtwhA`
- `http://is.gd/rPQj5E`

# Revision Control

```
$ cd freebsd/bin/ls
$ git log | cat
```

# Links

## Revision Control:

`http://cvsbook.red-bean.com/cvsbook.html`

`http://svnbook.red-bean.com/`

`http://git-scm.com/`

## GDB:

`http://sources.redhat.com/gdb/current/onlinedocs/gdb_toc.html`

`http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Debug.html`

`http://www.unknownroad.com/rtfm/gdbtut/gdbtoc.html`